目录

介绍

SDK安装

编译工具链

调试工具集

rootfs和boot

BST-OS SDK用户指南

文档修订记录

版本号	修订记录	修订日期	作成者
V0.0.1	初稿作成	2020/08/28	丁宁 孙彪 康瑞强 杨程
V0.0.2	docker.md	2020/09/09	杨程
V0.0.3	docker.md，Images.md	2020/09/11	杨程
V0.0.4	docker.md	2020/09/14	杨程
V0.8.0	rootfs boot更新	2020/09/15	杨程
V0.8.2	docker 提供串口连接内容	2020/09/21	杨程
V0.8.4	docker 不提供用户提交为新镜像的方法	2020/09/27	杨程
V0.8.6	更新SDK镜像包 版本号和获取路径	2020/10/12	杨程
V0.8.7	更新adb 内容	2020/10/20	杨程
V0.8.8	更新rootfs boot 内容	2020/10/30	杨程
V0.8.9	删除 docker vnc连接相关内容	2021/01/20	杨程
V0.9.0	新增版本包和镜像包单板类型区分	2021/03/12	杨程

内容提要

BST-OS SDK是一套面向BST-OS开发和使用者的开发工具包，其基于黑芝麻智能科技操作系统平台BST-OS，为开发者提供完整的API接口、功能强大的工具套件和丰富的应用实例。BST-OS SDK旨在帮助开发者专注于应用本身，快速、高效的开发和移植应用软件。

本文为您介绍BST-OS SDK的详细信息，包括下载、安装、更新、使用、调试、示例等内容。

术语表

术语或缩略语	描述
BST-OS	Black Sesame Technologies Operating System 缩写，即黑芝麻智能科技操作系统，由黑芝麻智能科技提供。
SDK	Software Development Kit, 软件开发工具包

环境要求

在安装和使用BST-OS SDK前，需确认本地环境能否满足以下要求。

硬件

一台Linux x86-64的电脑，硬件要求如下：

（1）CPU i5及以上；

（2）内存不小于4G；

（3）硬盘可用空间不小于100GB。

软件

（1）推荐使用ubuntu18.04及以上操作系统；

SDK Docker部署

提供Docker Image方式部署SDK，部署成功后，编译操作在Docker Image中进行，提供镜像更新操作。

安装Docker

部署SDK Docker Image之前需要在本地安装Docker。

Docker 是一个开源的应用容器引擎，让开发者可以打包他们的应用以及依赖包到一个可移植的镜像中，然后发布到任何流行的 Linux或Windows 机器上，也可以实现虚拟化。容器是完全使用沙箱机制，相互之间不会有任何接口。

docker-ce下载地址：https://mirrors.tuna.tsinghua.edu.cn/docker-ce/linux/

举例：

示例包下载地址：

https://mirrors.tuna.tsinghua.edu.cn/docker-ce/linux/ubuntu/dists/bionic/pool/stable/amd64/

https://ubuntu.pkgs.org/18.04/ubuntu-main-amd64/libltdl7_2.4.6-2_amd64.deb.html

环境信息：操作系统：ubuntu18.04，docker-ce软件版本 docker-ce_18.09

```
root@ubuntu:~$ ls -l
docker-ce-cli_18.09.6~3-0~ubuntu-bionic_amd64.deb
containerd.io_1.2.5-1_amd64.deb
docker-ce_18.09.6~3-0~ubuntu-bionic_amd64.deb
libltdl7_2.4.6-2_amd64.deb
```

```
root@ubuntu:~$ sudo dpkg -i libltdl7_2.4.6-2_amd64.deb
root@ubuntu:~$ sudo dpkg -i docker-ce-cli_18.09.6~3-0~ubuntu-bionic_amd64.deb
root@ubuntu:~$ sudo dpkg -i containerd.io_1.2.5-1_amd64.deb
root@ubuntu:~$ sudo dpkg -i docker-ce_18.09.6~3-0~ubuntu-bionic_amd64.deb
root@ubrootuntu:~$ docker -v
Docker version 18.09.6, build 481bc77
```

下载SDK-Docker包

下载地址 http://dev.bstai.top/

${version} 当前最新版本号

${board_type} 当前设备类型

下载BST-HS-Linux-SDK ${version}.zip并解压,其中包括SDK和Image完整包

SDK镜像包路径如下: SDK-Docker-${version}.zip

```
root@ubuntu:~$sdk-a1000-docker$ unzip BST-HS-Linux-SDK-${board_type}-${version}.zip
BST-HS-Linux-SDK ${version}
root@ubuntu:~$sdk-a1000-docker$ ls BST-HS-Linux-SDK-${board_type}-${version}/sdk/sdk-docker-build/ &&cd BST-HS-Linux-SDK
SDK-Docker-${board_type}-${version}.zip
root@ubuntu:~$sdk-docker-build$ unzip SDK-Docker-${board_type}-${version}.zip && cd SDK-Docker-${board_type}-${version}/
a1000-sdk-${version}.tgz  run_docker.sh
```

运行**Docker Image**

Run docker步骤:

注意：

run_docker.sh每次运行执行都会清理并删除上次已经运行起来的容器环境，并把容器环境端口5900映射到宿主机端口5900，请保证宿主机5900端口不被占用；

执行run_docker.sh 时会提示安装pigz解压缩工具，有网环境下可使用apt-get install pigz 直接安装；

adb工具不支持 dockers环境和宿主机环境同时启动，需执行adb kill-server 停止宿主机adb服务；

等待镜像导入并生成容器:

```
root@ubuntu:~SDK-Docke$chmod +x ./run_docker.sh && sudo ./run_docker.sh
running  docker env...

e1c4eb6252e50fd2ea4812c4a03ebee58116fbad240a7e5565613d488f91948c3

root@e1c4eb6252e5:/home#
```

SSH连接**docker**容器**SDK**环境

前提准备：

一台支持桌面的主机，docker容器宿主机，在同一网段，ssh终端连接软件。

注意：docker镜像环境默认提供 ssh 端口 212 用于远程连接。

使用示例：

win10, mobaxterm 12.4,ubuntu18.04系统环境，SDK docker镜像,默认提供端口212，用户以及密码：root/123456；

使用mobaxterm软件，创建ssh连接 ，ip: 提供ubuntu18.04系统环境 网络ip，端口号，用户。

使用串口连接设备

SDK docker镜像为用户提供使用工具链内核调试工具，通过串口连接到A1000设备。

前提准备：

pc机，SDK docker镜像 ,usb线，A1000设备

注意：

pc机如果是KVM则需要对应物理机接入USB设备映射到KVM上并能正确获取设备名称，docker环境即可使用对应设备；

设备串口不支持同时使用相同USB设备名连接docker环境和宿主机环境，若要在Docker环境中连接设备串口，需要确保设备串口端口在Docker宿主机中未被使用。

使用示例：

ubtun 18.04 64 pc机,usb接口线，A1000设备

通过usb接口线连接pc机和A1000设备，会显示该设备名称

```
root@ubuntu:~$sdk-a1000-docker$ ls /dev/tty*
ttyUSBXXX
```

保存**docker**容器内容变更

保存容器内容步骤前提：

获取容器id,当执行run_docker.sh脚本后id如下；

注意：e31b46716cb6 每次生成容器均会重新生成此id，

```
root@ubuntu:~SDK-Docke$sudo ./run_docker.sh
running start docker env...

e31b46716cb60fd2ea4812c4a03ebee58116fbad240a7e5565613d488f91948c3

root@e31b46716cb6:/home#
```

执行命令如下：

在当前运行的docker容器进行需要内容添加删除修改；

测试示例：

提交修改到默认镜像中（a1000-sdk:${version}）

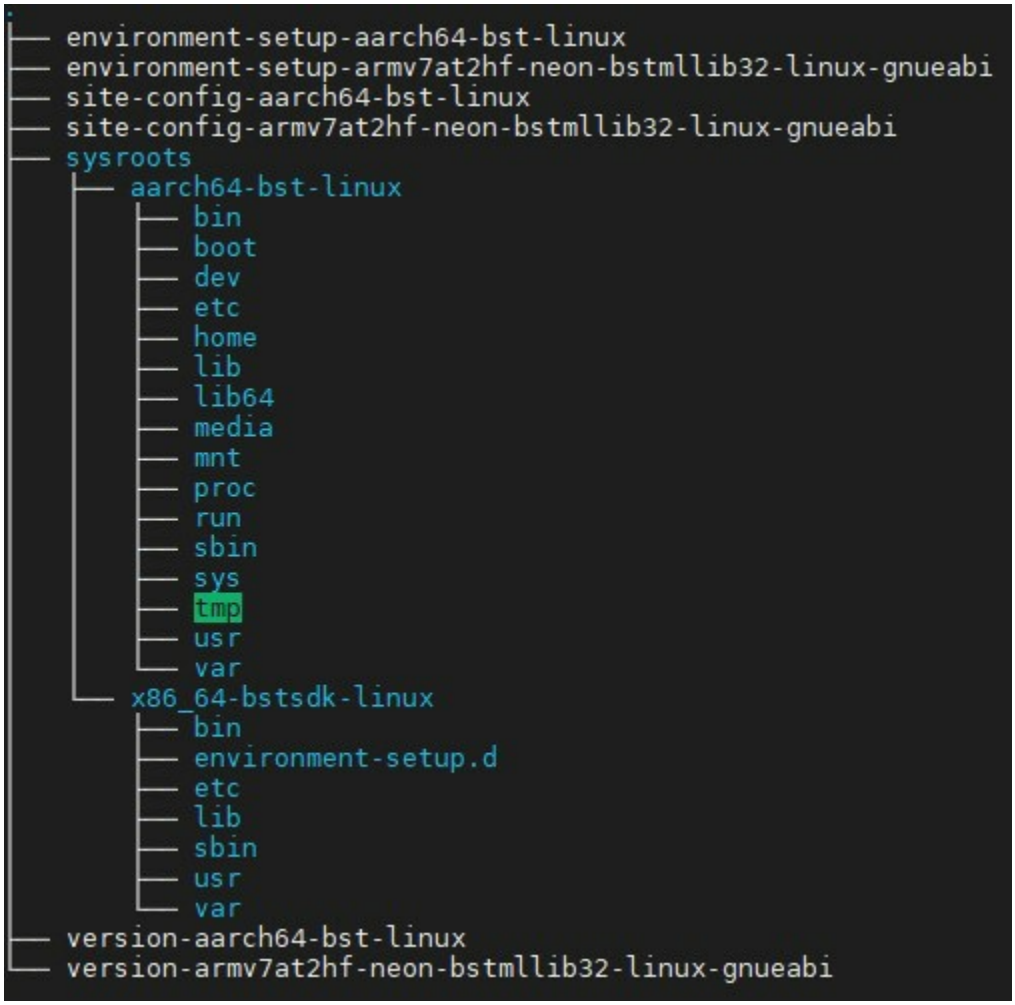
注意：提交修改到默认镜像a1000-sdk:${version}，可继续使用./run_docker.sh运行容器进行功能支持；

```
root@e31b46716cb6:/sdk#touch filetest
root@e31b46716cb6:/sdk#exit
#更新镜像
root@dell-PowerEdge-R740:$sudo docker commit  -a "author" -m "add new filetest" e31b46716cb6 a1000-sdk:${version}
root@dell-PowerEdge-R740:$ sudo docker images |grep a1000-sdk
a1000-sdk     ${version}   18c715396529 1 minutes ago      7.83GB
root@dell-PowerEdge-R740:$sudo ./run_docker.sh
running  docker env...

e1c4eb6252e50fd2ea4812c4a03ebee58116fbad240a7e5565613d488f91948c3
root@edf446716cb6:/sdk#ls
filetest
```

文件结构概览

BST-OS SDK工具包结构如下图所示：



文件说明

文件或目录	描述
environment-setup-aarch64-bst-linux	SDK 64位环境变量
site-config-aarch64-bst-linux	目标环境的配置文件
sysroots	适用于目标系统的应用开发所需要的文件系统
version-aarch64-bst-linux	版本信息
environment-setup-armv7at2hf-neon-bstmllib32-linux-gnueabi	SDK 32位环境变量
site-config-armv7at2hf-neon-bstmllib32-linux-gnueabi	目标环境的配置文件

version-armv7at2hf-neon-bstmllib32-linux-gnueabi	版本信息

以上文件在一般情况下无需修改，特殊需求需咨询开发人员。

编译工具链介绍

本章主要介绍BST-OS SDK中已集成的常用编译工具及其版本、典型应用等内容。

使用SDK中调试工具时对应环境变量内容如下，SDK默认已配置完成：

```
/{install_path}/environment-setup-aarch64-bst-linux
```

{install_path}表示BST-OS SDK安装路径。

gcc-g++

GNU Compiler Collection，GNU编译器套件

GNU编译器套件包括gcc,g++

集成版本

```
aarch64-bst-linux-gcc (GCC) 8.3.0
aarch64-bst-linux-g++ (GCC) 8.3.0
aarch64-bst-linux-gfortran (GCC) 8.3.0
```

所在位置

```
which aarch64-bst-linux-gcc
/{install_path}/sysroots/x86_64-bstsdk-linux/usr/bin/aarch64-bst-linux/aarch64-bst-linux-gcc

which aarch64-bst-linux-g++
/{install_path}/sysroots/x86_64-bstsdk-linux/usr/bin/aarch64-bst-linux/aarch64-bst-linux-g++

which aarch64-bst-linux-gfortran
/{install_path}/sysroots/x86_64-bstsdk-linux/usr/bin/aarch64-bst-linux/aarch64-bst-linux-gfortran
```

使用方式

gcc/g++/gfortran 一般不直接使用 一般使用CMake构建方式，请参考cmake

配置说明

设置sdk环境变量时，会设置以下配置

```
declare -x CFLAGS=" -O2 -pipe -g -feliminate-unused-debug-types "
declare -x CONFIGURE_FLAGS="--target=aarch64-bst-linux --host=aarch64-bst-linux --build=x86_64-linux \
--with-libtool-sysroot=/{install_path}/sysroots/aarch64-bst-linux"
declare -x CPPFLAGS=""
declare -x CROSS_COMPILE="aarch64-bst-linux-"
declare -x CXXFLAGS=" -O2 -pipe -g -feliminate-unused-debug-types "
declare -x DISPLAY="localhost:19.0"
declare -x KCFLAGS="--sysroot=/{install_path}/sysroots/aarch64-bst-linux"
declare -x LANG="zh_CN.UTF-8"
declare -x LANGUAGE="zh_CN:zh"
declare -x LD="aarch64-bst-linux-ld  --sysroot=/{install_path}/sysroots/aarch64-bst-linux"
declare -x LDFLAGS="-Wl,-O1 -Wl,--hash-style=gnu -Wl,--as-needed"
declare -x OECORE_ACLOCAL_OPTS="-I /{install_path}/sysroots/x86_64-bstsdk-linux/usr/share/aclocal"
declare -x OECORE_BASELIB="lib"
declare -x OE_CMAKE_FIND_LIBRARY_CUSTOM_LIB_SUFFIX=""
declare -x OE_CMAKE_TOOLCHAIN_FILE=\
"/{install_path}/sysroots/x86_64-bstsdk-linux/usr/share/cmake/OEToolchainConfig.cmake"
declare -x OPENSSL_CONF="/{install_path}/sysroots/x86_64-bstsdk-linux/usr/lib/ssl-1.1/openssl.cnf"
```

```
declare -x RANLIB="aarch64-bst-linux-ranlib"
```

gfortran 示例

main.f 文件内容

```
      PRINT *, "Hello World!"
            END
```

测试步骤

打开终端

```
#set env
. /{install_path}/environment-setup-aarch64-bst-linux

#编译 其中install_path为sdk安装根目录
aarch64-bst-linux-gfortran main.f -o test --sysroot=/{install_path}/sysroots/aarch64-bst-linux

#拷贝可执行文件test到target端
chmod +x test
./test
```

注意：如果使用cmake编译fortran，需要添加环境变量

```
#其中install_path为sdk安装根目录
export FC="aarch64-bst-linux-gfortran  --sysroot=/{install_path}/sysroots/aarch64-bst-linux"
```

clang

TBD

CMake

CMake是一个跨平台的安装（编译）工具，可以用简单的语句来描述所有平台的安装(编译过程)。他能够输出各种各样的makefile或者project文件，能测试编译器所支持的C++特性，类似UNIX下的automake。关于CMake的详细信息及使用教程请参考CMake官网。

BST-OS SDK中已集成CMake工具，供开发者使用。

集成版本

```
cmake version 3.14.1
```

所在位置

```
$ which cmake
/{install_path}/sysroots/x86_64-bstsdk-linux/usr/bin/cmake
```

示例

CMakeLists.txt

```
project(test)

add_executable(test main.cpp)
```

main.cpp

```
#include <iostream>


int main(int argc, char* argv[])
{
    std::cout << "hello world!" << std::endl;
    return 0;
}
```

测试步骤

打开终端，进入CMakeLists.txt目录

```
mkdir build
cd build

#set env
. /{install_path}/environment-setup-aarch64-bst-linux
#configure
cmake ..
```

```
#compile
make

#拷贝可执行文件test到target端
chmod +x test
./test
```

python

BSTOS SDK已预先集成python3。

集成版本

python3.7

所在位置

```
$ which python3.7
/{install_path}/sysroots/x86_64-bstsdk-linux/usr/bin/python3.7
```

protoc

Google Protocol Buffer(简称 Protobuf) 是 Google 公司内部的混合语言数据标准。他们用于 RPC 系统和持续数据存储系统。提供一个具有高效的协议数据交换格式工具库(类似Json)。 但相比于Json，Protobuf有更高的转化效率，时间效率和空间效率都是JSON的3-5倍。 可用于通讯协议、数据存储等领域的语言无关、平台无关、可扩展的序列化结构数据格式。Protobuf文件以.proto为后缀，有自己的编译器 protoc, protoc2 和 protoc3 版本。

详细信息及教程可参考Google官方.protoc文档。

BSTOS SDK已预先集成protoc3。

集成版本

libprotoc 3.6.1

所在位置

```
$ which protoc
/{install_path}/sysroots/x86_64-bstsdk-linux/usr/bin/protoc
```

编译文件

使用protoc编译protobuf文件需要先写.proto，假设需要编译存放在 $SRC_DIR 路径下的simple.proto文件，$DST_DIR目录为生成的文件目录，则可以使用如下命令:

```
protoc --proto_path=$SRC_DIR --cpp_out=$DST_DIR src/main/proto/simple.proto
```

nanolog

操作系统运行过程当中会产生许多信息，这些信息既是我们观察系统运行过程当中正常的一种途径，同时它也为我们提供了当发生故障时定位问题所在的必要信息。

nanolog以C++开发，对nanolog使用者提供一份接口头文件NanoLogCpp17.h。

API及功能简要描述

`NANO_LOG(LogLevels, , , )`

记录日志的等级以及需要记录的日志消息。

`void preallocate()`

为当前线程预先分配日志系统所需的线程本地数据结构。尽管是可选的，但是建议在每个将要使用日志系统的线程中在第一个日志消息之前调用这个函数。

`void setLogFile(const char* filename)`

设置NanoLog输出的文件位置。在这个函数返回后调用的所有NANO_LOG语句都保证位于新文件的位置

`void setLogLevel(LogLevel logLevel)`

设置系统中的最小日志严重级别。所有日志严重程度较低的日志语句将被完全删除。

`LogLevel getLogLevel()`

获取系统设置的最小日志严重级别。

`void sync()`

等待，直到所有待处理的日志语句都存储到磁盘上。 注意，如果还有另一个日志记录线程不断添加新的挂起日志语句，直到所有线程停止日志记录以及所有新的日志语句存储到磁盘这个函数才会返回。

调用日志系统

在需要记录日志的CPP文件中包含NanoLog接口头文件NanoLogCpp17.h

将NANO_LOG（）嵌入到需要记录的代码段中

调用其他的API对日志进行设置

编译时将libNanoLog.a静态链接到可执行程序中

在运行可执行程序时，日志系统就会记录相应的日志，并将其压缩保存到相应存储介质

查看日志信息

通过执行 decompressor decompress ${日志存储位置}即可查看日志。目前，decompressor已经集成到系统中，后续会将其集成到系统的IDE中。

开发示例

Demo.cpp

```
#include <chrono>

// Required to use the NanoLog system
#include "Nanolog/NanoLogCpp17.h"

void runBenchmark(int num);
//void runBenchmark();

// Optional: Import the NanoLog log levels into the current namespace; this
// allows the log levels (DEBUG, NOTICE, WARNING, ERROR) to be used without
// using the NanoLog namespace (i.e. NanoLog::DEBUG).
using namespace NanoLog::LogLevels;

int main(int argc, char** argv) {
    // Optional: Set the output location for the NanoLog system. By default
    // the log will be output to ./compressedLog
    NanoLog::setLogFile("/tmp/logFile");

    // Optional optimization: pre-allocates thread-local data structures
    // needed by NanoLog. This can be invoked once per new
    // thread that will use the NanoLog system.
    NanoLog::preallocate();

    // Optional: Set the minimum LogLevel that log messages must have to be
    // persisted. Valid from least to greatest values are
    // DEBUG, NOTICE, WARNING, ERROR
    NanoLog::setLogLevel(NOTICE);

    NANO_LOG(DEBUG, "This message wont be logged since it is lower "
                        "than the current log level.");

    NANO_LOG(DEBUG, "Another message.");

    // All the standard printf specifiers (except %n) can be used
    char randomString[] = "Hello World";
    NANO_LOG(NOTICE, "A string, pointer, number, and float: '%s', %p, %d, %f",
                        randomString,
                        &randomString,
                        512,
                        3.14159);

    // Even with width and length specifiers
    NANO_LOG(NOTICE, "Shortend String: '%5s' and shortend float %0.2lf",
                    randomString,
                    3.14159);

    uint64_t num = atol(argv[1]);
    runBenchmark(num);

    // Optional: Flush all pending log messages to disk
    NanoLog::sync();

    // Optional: Gather statics generated by NanoLog
    std::string stats = NanoLog::getStats();
    printf("%s", stats.c_str());

    // Optional: Prints NanoLog configuration parameters
    NanoLog::printConfig();
}

void runBenchmark(int num) {

    uint64_t RECORDS = num;

    std::chrono::high_resolution_clock::time_point start, stop;
```

```
    double time_span;

    start = std::chrono::high_resolution_clock::now();
    for (int i = 0; i < RECORDS; ++i) {
        NANO_LOG(NOTICE, "Simple log message with 0 parameters");
    }
    stop = std::chrono::high_resolution_clock::now();

    time_span = std::chrono::duration_cast<std::chrono::duration<double>>(
                                                    stop - start).count();
    printf("The total time spent invoking NANO_LOG with no parameters %lu "
           "times took %0.2lf seconds (%0.2lf ns/message average)\r\n",
           RECORDS, time_span, (time_span/RECORDS)*1e9);

    start = std::chrono::high_resolution_clock::now();
    // Flush all pending log messages to disk
    NanoLog::sync();
    stop = std::chrono::high_resolution_clock::now();

    time_span = std::chrono::duration_cast<std::chrono::duration<double>>(
                                                    stop - start).count();
    printf("Flushing the log statements to disk took an additional "
                "%0.2lf secs\r\n", time_span);
}
```

CMakeList.txt

```
project(nano-test)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)

set(CMAKE_C_FLAGS_Release "${CMAKE_C_FLAGS_Release} -O3 -Wall")
SET(CMAKE_BUILD_TYPE "Release")

add_executable(test main.cpp)
target_link_libraries(test libNanoLog.a pthread rt)

install(TARGETS test)
```

调试工具集

本章主要介绍BST-OS SDK中已集成的常用调试工具及其版本等内容。

使用SDK中调试工具时对应环境变量内容如下，SDK默认已配置完成：

```
/{install_path}/environment-setup-aarch64-bst-linux
```

{install_path}表示BST-OS SDK安装路径。

- fastboot

fastboot

fastboot是一种需要通过数据线连接设备进行刷机的工具。

本章仅列举fastboot常用操作命令，详细教程请参考：fastboot教程

sdk docker中默认已集成fastboot工具

Docker端操作

烧写开始前，需提前下载image文件，下载BST-HS-Linux-SDK ${version}.zip并解压,其中包括SDK和Image完整包。

下载地址 http://dev.bstai.top/

（1）烧写分区表

```
fastboot flash gpt partition.img
```

（2）烧写boot分区

```
fastboot flash boot boot.img
```

（3）烧写persistence分区

```
fastboot flash persistence  persistence.img
```

（4）烧写rootfs分区

```
fastboot flash rootfs core-image-bstos.img
```

（5）如果配置了bootcmd,下面语句会执行bootcmd

```
fastboot continue
```

adb

adb 是一种功能多样的命令行工具，可让您与设备进行通信。adb 命令可用于执行各种设备操作（例如安装和调试应用），并提供对 Unix shell（可用来在设备上运行各种命令）的访问权限。

本章仅列举adb常用操作命令，详细教程请参考：adb教程

使用adb

sdk docker中默认已集成adb工具，可在docker环境执行adb命令。

开始使用前确认PC与A1000之间usb线的已连接，且PC主机上adb服务需要执行adb kill-server避免与docker容器中冲突。

文件或目录	描述
adb push $SRC_DIR/$fileName /$DST_DIR	将本地$SRC_DIR目录下文件$fileName推送到A1000的$DST_DIR目录下
adb pull $SRC_DIR/$fileName /$DST_DIR	将A1000$SRC_DIR目录下文件$fileName拉取到本地$DST_DIR目录下
adb shell	进入A1000 shell中
adb shell dmesg	打印kernel log

$SRC_DIR:原目录

$DST_DIR:目标目录

$fileName:文件名称

Logviewer

TBD

perf

集成版本perf版本V1.0。Perf 是内置于Linux 内核源码树中的性能剖析（profiling）工具。它基于事件采样原理，以性能事件为基础，支持针对处理器相关性能指标与操作系统相关性能指标的性能剖析。可用于性能瓶颈的查找与热点代码的定位。linux2.6及后续版本都自带该工具，几乎能够处理所有与性能相关的事件。关于Perf详细介绍请参考：Perf简介 , Perf常用命令

Perf是一个包含22种子工具的工具集，以下是最常用的5种。

perf-list

Perf-list用来查看perf所支持的性能事件，有软件的也有硬件的。

List all symbolic event types.

perf list [hw | sw | cache | tracepoint | event_glob]

1. 性能事件的分布

hw：Hardware event，9个

sw：Software event，9个

cache：Hardware cache event，26个

tracepoint：Tracepoint event，775个

sw实际上是内核的计数器，与硬件无关。

hw和cache是CPU架构相关的，依赖于具体硬件。

tracepoint是基于内核的ftrace，主线2.6.3x以上的内核版本才支持。

1. 指定性能事件

-e : u // userspace

-e : k // kernel

-e : h // hypervisor

-e : G // guest counting (in KVM guests)

-e : H // host counting (not in KVM guests)

1. 使用例子

显示内核和模块中，消耗最多CPU周期的函数：

```
# perf top -e cycles:k
```

perf-stat

用于分析指定程序的性能概况。

Run a command and gather performance counter statistics.

perf stat [-e | --event=EVENT] [-a]

perf stat [-e | --event=EVENT] [-a] - []

1. 输出格式

```
# perf stat ls
```

```
Performance counter stats for 'ls':

          0.653782 task-clock                #    0.691 CPUs utilized
                 0 context-switches          #    0.000 K/sec
                 0 CPU-migrations            #    0.000 K/sec
               247 page-faults               #    0.378 M/sec
         1,625,426 cycles                    #    2.486 GHz
         1,050,293 stalled-cycles-frontend   #   64.62% frontend cycles idle
           838,781 stalled-cycles-backend    #   51.60% backend  cycles idle
         1,055,735 instructions              #    0.65  insns per cycle
                                             #    0.99  stalled cycles per insn
           210,587 branches                  #  322.106 M/sec
            10,809 branch-misses             #    5.13% of all branches

       0.000945883 seconds time elapsed
```

输出包括ls的执行时间，以及10个性能事件的统计。

项目	描述
task-clock	任务真正占用的处理器时间，单位为ms。CPUs utilized = task-clock / time elapsed，CPU的占用率。
context-switches	上下文的切换次数
CPU-migrations	处理器迁移次数
page-faults	缺页异常的次数
cycles	消耗的处理器周期数
stalled-cycles-frontend	指令读取或解码的指令步骤
stalled-cycles-backend	指令执行步骤
instructions	执行了多少条指令。IPC为平均每个cpu cycle执行了多少条指令。
branches	遇到的分支指令数。
branch-misses	预测错误的分支指令数

1. 常用参数

-p：stat events on existing process id (comma separated list). 仅分析目标进程及其创建的线程。

-a：system-wide collection from all CPUs. 从所有CPU上收集性能数据。

-r：repeat command and print average + stddev (max: 100). 重复执行命令求平均。

-C：Count only on the list of CPUs provided (comma separated list), 从指定CPU上收集性能数据。

-v：be more verbose (show counter open errors, etc), 显示更多性能数据。

-n：null run - don't start any counters，只显示任务的执行时间 。

-x SEP：指定输出列的分隔符。

-o file：指定输出文件，--append指定追加模式。

--pre ：执行目标程序前先执行的程序。

--post ：执行目标程序后再执行的程序。

1. 使用例子

执行10次程序，给出标准偏差与期望的比值：

```
$perf stat -r 10 ls > /dev/null
```

显示更详细的信息：

```
perf stat -v ls > /dev/null
$perf stat -v ls > /dev/null
```

只显示任务执行时间，不显示性能计数器：

```
$ perf stat -n ls > /dev/null
```

单独给出每个CPU上的信息：

```
$ perf stat -a -A ls > /dev/null
```

perf-top

对于一个指定的性能事件(默认是CPU周期)，显示消耗最多的函数或指令。

perf top [-e | --event=EVENT] []

perf top主要用于实时分析各个函数在某个性能事件上的热度，能够快速的定位热点函数，包括应用程序函数、模块函数与内核函数，甚至能够定位到热点指令。默认的性能事件为cpu cycles。

1. 输出格式

```
# perf top
```

```
Samples: 1M of event 'cycles', Event count (approx.): 73891391490
     5.44%  perf              [.] 0x0000000000023256
     4.86%  [kernel]          [k] _spin_lock
     2.43%  [kernel]          [k] _spin_lock_bh
     2.29%  [kernel]          [k] _spin_lock_irqsave
     1.77%  [kernel]          [k] __d_lookup
     1.55%  libc-2.12.so      [.] __strcmp_sse42
     1.43%  nginx             [.] ngx_vslprintf
     1.37%  [kernel]          [k] tcp_poll
```

第一列：符号引发的性能事件的比例，默认指占用的cpu周期比例。

第二列：符号所在的DSO(Dynamic Shared Object)，可以是应用程序、内核、动态链接库、模块。

第三列：DSO的类型。[.]表示此符号属于用户态的ELF文件，包括可执行文件与动态链接库)。[k]表述此符号属于内核或模块。

第四列：符号名。有些符号不能解析为函数名，只能用地址表示。

1. 常用交互命令

h：显示帮助

UP/DOWN/PGUP/PGDN/SPACE：上下和翻页。

a：annotate current symbol，注解当前符号。能够给出汇编语言的注解，给出各条指令的采样率。

d：过滤掉所有不属于此DSO的符号。非常方便查看同一类别的符号。

P：将当前信息保存到perf.hist.N中。

1. 常用命令行参数

-e ：指明要分析的性能事件。

-p ：Profile events on existing Process ID (comma sperated list). 仅分析目标进程及其创建的线程。

-k ：Path to vmlinux. Required for annotation functionality. 带符号表的内核映像所在的路径。

-K：不显示属于内核或模块的符号。

-U：不显示属于用户态程序的符号。

-d ：界面的刷新周期，默认为2s，因为perf top默认每2s从mmap的内存区域读取一次性能数据。

1. 使用例子

/默认配置

```
perf top
```

指定性能事件

```
perf top -e cycles
```

查看这两个进程的cpu cycles使用情况

```
perf top -p 23015,32476
```

显示调用symbol的进程名和进程号

```
perf top -s comm,pid,symbol
```

仅显示属于指定进程的符号

```
perf top --comms nginx,top
```

显示指定的符号

```
perf top --symbols kfree
```

perf-record

收集采样信息，并将其记录在数据文件中。

随后可以通过其它工具(perf-report)对数据文件进行分析，结果类似于perf-top的。

1. 常用参数

-e：Select the PMU event.

-a：System-wide collection from all CPUs.

-p：Record events on existing process ID (comma separated list).

-A：Append to the output file to do incremental profiling.

-f：Overwrite existing data file.

-o：Output file name.

-g：Do call-graph (stack chain/backtrace) recording.

-C：Collect samples only on the list of CPUs provided.

1. 使用例子

记录nginx进程的性能数据：

```
$ perf record -p `pgrep -d ',' nginx`
```

记录执行ls时的性能数据：

```
$ perf record ls -g
```

perf-report

读取perf record创建的数据文件，并给出热点分析结果。

1. 常用参数

-i：Input file name. (default: perf.data)

1. 使用例子

```
$ perf report -i perf.data.2
```

Valgrind

Valgrind是一个GPL的软件，用于Linux（For x86, amd64 and ppc32）程序的内存调试和代码剖析。你可以在它的环境中运行你的程序来监视内存的使用情况，比如C 语言中的malloc和free或者 C++中的new和 delete。使用Valgrind的工具包，你可以自动的检测许多内存管理和线程的bug，避免花费太多的时间在bug寻找上，使得你的程序更加稳固。详细说明和教程请参考：Valgrind User Manual

集成版本

3.14.0

工具集

Valgrind是一个用于构建动态分析工具的工具框架。它附带了一组工具，每个工具执行某种调试、分析或类似的任务，以帮助您改进程序。Valgrind的架构是模块化的，因此可以轻松创建新工具，而不会干扰现有的结构。

工具	描述
Memcheck	检查程序中的内存问题，如泄漏、越界、非法指针等
Cachegrind	分析CPU的cache命中率、丢失率，用于进行代码优化
callgrind	检测程序代码覆盖，以及分析程序性能
Helgrind	线程错误检查器，检查多线程程序的竞态条件
DRD	另一个线程错误检查器，和Helgrind类似，但是使用不同的分析技术
Massif	堆栈分析器，指示程序中使用了多少堆内存等信息
DHAT	一种不同类型的堆栈分析器，它帮助您理解块生存期、块利用率和布局低效等问题。
BBV	一个实验性的SimPoint基本块矢量生成器。它对进行计算机体系结构研究和开发的人很有用。
Lackey	个示例工具，用于说明一些仪器基础知识。
Nulgrind	最小的Valgrind工具，不进行分析或检测，仅用于测试目的。

Blktrace

Blktrace是一个针对Linux内核中块设备I/O层的跟踪工具，用来收集磁盘IO信息中当IO进行到块设备层（block层，所以叫blk trace）时的详细信息（如IO请求提交，入队，合并，完成等等一些列的信息）。通过使用这个工具，使用者可以获取I/O请求队列的各种详细的情况，包括进行读写的进程名称、进程号、执行时间、读写的物理块号、块大小等等。

具体使用方法可参考：blktrace(8) — Linux manual page

具体使用教程总结:硬盘读写追踪工具Blktrace&Blkparse

集成版本

blktrace 2.0.0

常用方法

使用blktrace需要挂载debugfs：

```
$ mount -t debugfs debugfs /sys/kernel/debug
```

利用blktrace查看实时数据的方法，比如要看的硬盘是sdb（需要停止的时候，按Ctrl-C）

```
$ blktrace -d /dev/sdb -o - | blkparse -i -
```

以上命令也可以用下面的脚本代替

```
$ btrace /dev/sdb
```

利用blktrace把数据记录在文件里，以供事后分析，缺省的输出文件名是 sdb.blktrace.，每个CPU对应一个文件,你也可以用-o参数指定自己的输出文件名。

```
$ blktrace -d /dev/sdb
```

利用blkparse命令分析blktrace记录的数据：

```
$ blkparse -i sdb
```

Img解压缩

对rootfs.img boot.img文件解压、压缩，img内容新增修改删除。

重新制作rootfs

提供rootfs.img 内容新增修改删除方式。

注意：运行环境不区分容器环境或者pc机，只需安装对应工具即可。

安装make_ext4fs simg2img

安装make_ext4,simg2img工具

```
ubuntu@ubuntu:~$ sudo apt-get update
ubuntu@ubuntu:~$ sudo apt-get install -y android-tools-fsutils
```

文件解压

提前准备：

core-image-bstos.img文件

操作示例：

测试环境ubuntu 18.04 64位

准备一个 core-image-bstos.img文件

如果core-image-bstos.img是SDK中官方自带，操作如下：

```
ubuntu@ubuntu:~$sudo ls
core-image-bstos.img
ubuntu@ubuntu:~$sudo mkdir  fs
ubuntu@ubuntu:~$sudo simg2img core-image-bstos.img core-image-bstos.img.ext4
ubuntu@ubuntu:~$sudo mount -t ext4 -o loop core-image-bstos.img.ext4 ./fs
ubuntu@ubuntu:~$sudo ls ./fs
bin  boot  dev  etc  home  lib  lost+found  media  mnt  proc  run  sbin  sys  tmp  usr  var
```

如果core-image-bstos.img是已被simg2img转换为原始img并重新打包生成，操作如下：

```
ubuntu@ubuntu:~$sudo ls
core-image-bstos.img
ubuntu@ubuntu:~$sudo mkdir  fs
ubuntu@ubuntu:~$sudo mount -t ext4 -o loop core-image-bstos.img ./fs
ubuntu@ubuntu:~$sudo ls ./fs
bin  boot  dev  etc  home  lib  lost+found  media  mnt  proc  run  sbin  sys  tmp  usr  var
```

对文件内容新增修改删除

对img 内容进行变更

```
ubuntu@ubuntu:~$cd ./fs/home/
ubuntu@ubuntu:~$sudo touch filetest
```

文件重新打包

打包操作如下：

```
ubuntu@ubuntu:~$cd ../../ && mv core-image-bstos.img core-image-bstos.img-old && ls
fs    core-image-bstos.img-old
ubuntu@ubuntu:~$sudo make_ext4fs -l 4096M core-image-bstos.img  ./fs/
ubuntu@ubuntu:~$ls
fs core-image-bstos.img core-image-bstos.img-old
ubuntu@ubuntu:~$umount ./fs/ &&rm -rf ./fs/
```

如果觉得镜像文件过大，可以使用下面的命令将raw img转换为sparse image

img2simg使用说明：

Usage: img2simg raw_image_file sparse_image_file block_size

重新制作boot

提供boot.img 内容更新方式。

注意：运行环境不区分容器环境或者pc机，只需安装对应工具即可。

安装make_ext4fs simg2img

安装make_ext4,simg2img工具

```
ubuntu@ubuntu:~$ sudo apt-get update
ubuntu@ubuntu:~$ sudo apt-get install -y android-tools-fsutils
```

文件解压

提前准备：

boot.img文件

操作示例：

测试环境ubuntu 18.04 64位

准备一个boot.img文件

如果boot.img是SDK中官方自带，操作如下：

```
ubuntu@ubuntu:~$sudo ls
boot.img
ubuntu@ubuntu:~$sudo mkdir  fs
ubuntu@ubuntu:~$sudo simg2img boot.img boot.img.ext4
ubuntu@ubuntu:~$sudo mount -t ext4 -o loop boot.img.ext4 ./fs
ubuntu@ubuntu:~$sudo ls ./fs
Image.itb
```

如果boot.img是已被simg2img转换为原始img并重新打包生成，操作如下：

```
ubuntu@ubuntu:~$sudo ls
boot.img
ubuntu@ubuntu:~$sudo mkdir  fs
ubuntu@ubuntu:~$sudo mount -t ext4 -o loop boot.img ./fs
ubuntu@ubuntu:~$sudo ls ./fs
Image.itb
```

对文件内容更新

对img 内容进行变更,更新替换uImage bsta1000.dtb文件

```
ubuntu@ubuntu:~$cd ./fs/
ubuntu@ubuntu:~$ls
Image.itb
```

文件重新打包

打包操作如下：

```
ubuntu@ubuntu:~$cd ..  && mv boot.img boot.img-old && ls  #注意：把旧的boot.img 修改名称区别
fs    boot.img-old
ubuntu@ubuntu:~$sudo make_ext4fs -l 512M boot.img  ./fs/
ubuntu@ubuntu:~$ls
fs boot.img boot.img-old
ubuntu@ubuntu:~$umount ./fs/ &&rm -rf ./fs/
```

如果觉得镜像文件过大，可以使用下面的命令将raw img转换为sparse image

img2simg使用说明：

Usage: img2simg raw_image_file sparse_image_file block_size